Reflection

Reflection

At run-time, interrogate classes -- info about classes and methods
Not using source code -- just using Java's standard runtime state (from .class files)

Class

The "Class" class stores the state about a class
Methods
Constructors
IVars
newInstance() -- return a new instance of the class

Invoke

Given a string like "doit", find the matching Method object

Analysis

Significant
- A profoundly enabling feature -- a radical change from the C/C++ compile-them-together view of combining objects

Run-Time Combination
- Objects do not need to be compiled together -- can lookup and invoke methods that you did not know the name of at run-time.
- Makes it easier to get separate classes to work together
- Serialization and Beans (below) depend on Reflection

Underutilized
- I believe that 20 years of C/C++ program have trained us to underutilize the sort of flexibility that reflection enables. There are going to be big changes as we learn to utilize it.

Reflection/DB Example

Suppose you want to have Java objects in memory, but you want to save them to a database -- 1 column per ivar, 1 row per objects.
You could code the r/w translation by hand.
You could write reflection based code just once that can interrogate any java class enough to do the mapping automatically.

Reflection Button Example

Pre-Reflection Button
- In code, you wire up your listener to listen to the button and implement the message that it is coded to send.

- Required subclassing code

"Smart" Reflection Button

- Implement a "smart" reflection button that interacts with its environment depending on its state only -- no subclassing
- Edit the button **state**
- Put in a pointer to the object it should notify (state)
- Put in, as a string, the message that it should send (state)
- EG, suppose we have a Bell object that supports the ring() message.
- Can wire up an off the shelf smart button to send "ring()" at RT, and the Reflection figures it out.

Point: no code changes

- Can wire up the off-shelf smart button (no code changes) to the off-the-shelf Bell object.
- They work together, but with no subclassing, inner-classing etc. off either of them.
- All just done with **state, not code**

Point: client convenience

- Makes it easier to be a client
- Enables more buttons, bells, etc. that a client can just wire together
- State editing is much easier than code editing

Performance

- Java 1.2 GUI construction involves creating many little inner classes which creates significant overhead. Using reflection, although costly per-click, is a better use of resources.

Reflection Code

- See the example below

Class

- A "Class" object represents a java class

Method

- Represents a single method implemented in a Class

ReflectionDemo.java

```
// ReflectionDemo.java

import java.lang.reflect.*;
import java.io.*;

/*
 Uses Reflection to load, look at, and execute
 methods on classes at runtime.


 The code for this is gratefully adapted from...
 1) Jata in a Nutshell (www.ora.com/catalog/books/javanut2/)
 2) Bruce Eckel's book (www.bruceeckel.com)
*/

class ReflectionDemo extends Object {
```

```
// Find the Class object representing a class
// -- essentially calling the class loader
public static Class lookupClass(String className) {

   Class c = null;

   try {
      c = Class.forName(className);
   }
   catch (ClassNotFoundException e) {
      System.out.println(e.getMessage());
   }

   return(c);
}


// Print out info about a class
// (See the many accessors in the Class class)
public static void printClass(Class c) {
   System.out.println(Modifier.toString(c.getModifiers()) + " " +
      c.getName() +  " extends " + c.getSuperclass().getName());

   Field[] fields = c.getDeclaredFields();
   for (int i=0; i<fields.length; i++) {
      printField(fields[i]);
   }

   Constructor[] ctors = c.getDeclaredConstructors();
   for (int i=0; i<ctors.length; i++) {
      printMethod(ctors[i]);
   }

   Method[] methods = c.getDeclaredMethods();
   for (int i = 0; i<methods.length; i++) {
      printMethod(methods[i]);
   }
}

public static String typeToString(Class c) {
   String brackets = "";
   while (c.isArray()) {
      brackets += "[]";
      c = c.getComponentType();
   }
   return c.getName() + brackets;
}


public static void printField(Field field) {
   System.out.println(typeToString(field.getType()) + " " + field.getName());
}


public static void printMethod(Member member) {
   Class[] parameters;

   if (member instanceof Method) {
```

```
      Method method = (Method)member;
      parameters = method.getParameterTypes();
   }
   else {
      Constructor ctor = (Constructor)member;
      parameters = ctor.getParameterTypes();
   }

   System.out.print(member.getName());
   System.out.print(" (");
   for (int i=0; i<parameters.length; i++) {
      System.out.print(typeToString(parameters[i]) + " ");
   }
   System.out.println(")");
}

// Invoke a method with the given name (with no arguments)
public static void callNamedMethod(Class c, Object receiver, String methodName) {
   try  {
      Class[] metaArgs = new Class[0];

      Method method = c.getMethod(methodName, metaArgs);

      Object[] args = new Object[0];

      method.invoke(receiver, args);
   }
   catch (NoSuchMethodException e) {
      System.out.println("Err no method:" + e.getMessage());
   }
   catch (InvocationTargetException e) {
      System.out.println("Err target:" + e.getMessage());
   }
   catch (IllegalAccessException e) {
      System.out.println("Err access:" + e.getMessage());
   }
}


public static void userRun() {
   try {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader reader = new BufferedReader(isr);

      System.out.println("-----\nWhat class name?");

      String className = reader.readLine();

      if (className.equals("quit")) System.exit(0);

      System.out.println("Lookup: " + className);
      Class c = lookupClass(className);

      if (c==null) return;

      System.out.println("Info");
      printClass(c);
```

```
            System.out.println("\nNew instance");
            Object object = c.newInstance();

            System.out.println("What method?");
            String methodName = reader.readLine();

            System.out.println("Invoking: " + methodName);

            callNamedMethod(c, object, methodName);

         }
         catch (IOException e) {
            System.out.println("Err IO:" +e.getMessage());
         }
         catch (InstantiationException e) {
            System.out.println("Er instantiation:" +e.getMessage());
         }
         catch (IllegalAccessException e) {
            System.out.println("Err access:" +e.getMessage());
         }
      }


      public static void main(String[] args) {

         while(true) {
            userRun();
         }

      }
}
```

Foo.java

```
//Foo.java

// Just a silly class to use Reflection on

public class Foo {
   int a;
   String b;
   int[] ints;

   public Foo() {
      a = 13;
      b = "hello";
      ints = new int[100];
   }

   public void doit() {
      System.out.println("Who's calling me?");
   }

   public void doit(int x, String y) {
      a = a + 1;
   }
```

```
   private void bar() {
      a = a + 1;
   }
}
```

Output

```
What class name?
Foo
Lookup: Foo
Info
public synchronized Foo extends java.lang.Object
int a
java.lang.String b
int[] ints
Foo ()
doit ()
doit (int java.lang.String )
bar ()

New instance
What method?
doit
Invoking: doit
Who's calling me?
```

Java 1.4 -- New Serialization

http://java.sun.com/products/jfc/tsc/articles/persistence2/
GUI == a tree of components

XML Save GUI-Graph

Use a tool (bean editor) to edit and arrange visual components.
At runtime, "rehydrate" those objects into RAM

No: Listeners

Yes: Reflection connections

A control can have a couple bits of state: object wired to it + the message to send.
Key: record the wiring as state, not as a bunch of little listener classes

Code vs. State

Some aspects of a program are code oriented, and so editing those as code makes sense.
Other aspects are fundamentally state (button titles, component arrangement, strings, images, ...) and viewing and editing that state with a dedicated state editor is a huge improvement.